第2章

指令：计算机的语言

付俊宁 吕卫 译

指令集

- 一台计算机的全部指令
- 不同计算机有不同的指令集
  - 但在很多方面是相通的
- 早期计算机的指令集非常简单
  - 简化了实现过程
- 很多现代计算机同样具有简单的指令集

RISC-V指令集

- 用作示例贯穿全书
- 作为开放的ISA，由UC Berkeley开发
- 现由RISC-V基金会管理(riscv.org)
- 是众多现代ISA中的一个典型
  - 参见RISC-V参考数据卡
- 相近的ISA在嵌入式核心市场占据大量份额
  - 在消费电子、网络/存储设备、相机、打印机等等中的应用

算术运算指令

- 加法和减法，有三个操作数
  - 两个源操作数和一个目的操作数

```
add a, b, c   # b + c的结果保存到a
```

- 所有的算术运算指令都是这个格式
- **设计原则1：简单源于规整**
  - 规整的设计易于实现
  - 简单的设计能够以更低的成本获得更高的性能

算术运算的例子

- C代码：

```
f = (g + h) - (i + j);
```

- 编译后的RISC-V汇编代码：

```
add t0, g, h    # 临时变量t0 = g + h
add t1, i, j    # 临时变量t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

寄存器操作数

- 算术运算指令使用寄存器操作数
- RISC-V的寄存器文件由32个64位寄存器组成
  - 用于频繁访问的数据
  - 64位的数据称为“双字”
    - 32个64位通用寄存器编号从x0到x31
  - 32位的数据称为“字”
- 设计原则2：越小越快
  - 对比主存储器：数百万个地址

RISC-V寄存器

- x0: 常数0
- x1: 返回地址
- x2: 栈指针
- x3: 全局指针
- x4: 线程指针
- x5 – x7, x28 – x31: 临时变量
- x8: 帧指针
- x9, x18 – x27: 其值需要保存的寄存器
- x10 – x11: 函数参数/结果
- x12 – x17: 函数参数

寄存器操作数的例子

- C 代码：

```
f = (g + h) - (i + j);
```

  - f, …, j保存在x19, x20, …, x23中

- 编译后的RISC-V代码

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

存储器操作数

- 主存储器用于存储复合数据
  - 数组、结构、动态数据
- 为了进行算术运算操作
  - 把数值从存储器取到寄存器
  - 把结果从寄存器存到存储器
- 内存按字节编址
  - 每个地址对应一个8位字节
- RISC-V采用小端模式
  - 低位字节位于字中的低地址
  - 对比大端模式：高位字节位于低地址
- RISC-V不要求字在内存中对齐
  - 和其他一些ISA不同

存储器操作数的例子

- C代码：

  `A[12] = h + A[8];`

  - h保存在x21中，A的基址保存在x22中
- 编译后的RISC-V代码：
  - 下标8对应的偏移量是64
    - 每个双字有8个字节

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```

寄存器与存储器对比

- 访问寄存器比访问存储器快
- 对存储器数据进行操作需要取数和存数
  - 需要执行更多指令
- 编译器必须尽可能使用寄存器来保存变量
  - 只把不经常使用的变量转存到存储器
  - 寄存器优化很重要！

立即操作数

- 指令中指定的常数

```
addi x22, x22, 4
```

- 加速大概率事件
  - 小的常数是常见的
  - 立即数避免了load指令

无符号二进制整数

- 给定一个n位的数

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- 范围：从0到$+2^n - 1$
- 例
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- 使用64位
  - 从0到+18,446,774,073,709,551,615

二进制补码表示的有符号整数

- 给定一个n位的数

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- 范围：从-2^{n-1}到$+2^{n-1} - 1$
- 例(32位)
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- 使用64位：−9,223,372,036,854,775,808
  到 9,223,372,036,854,775,807

二进制补码表示的有符号整数

- 位63是符号位
  - 1表示负数
  - 0表示非负数
- 无法表示2^{n-1}
- 非负数的无符号数表示和二进制补码表示相同
- 几个特殊的数
  - 0 ： 0000 0000 … 0000
  - –1 ： 1111 1111 … 1111
  - 最小负数： 1000 0000 … 0000
  - 最大正数： 0111 1111 … 1111

有符号数取负

- 按位取反再加1
  - 取反指的是1 → 0、0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- 例：对+2取负
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

符号扩展

- 用更多的位表示一个数
  - 保持数值不变
- 把符号位复制到左边
  - 对比无符号数：用0扩展
- 例：把8位扩展为16位
  - +2：0000 0010 => 0000 0000 0000 0010
  - –2：1111 1110 => 1111 1111 1111 1110
- 在RISC-V指令集中
  - lb：符号扩展加载的字节
  - lbu：零扩展加载的字节

指令的表示

- 指令用二进制编码
  - 称为机器码
- RISC-V指令
  - 编码成32位的指令字
  - 用寥寥几种格式对操作码(opcode)、寄存器号等编码
  - 规整！

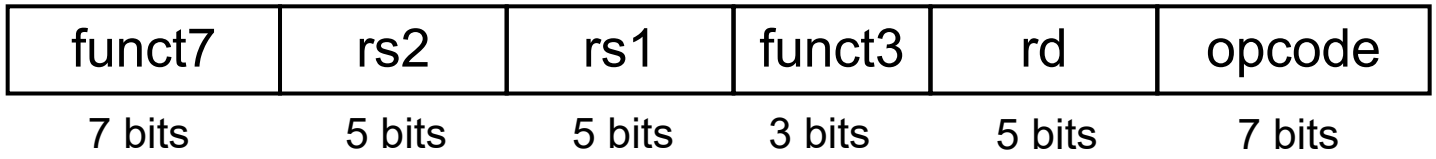
十六进制

- 基数为16
  - 位串的紧凑表示
  - 每位十六进制数对应4位二进制数

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

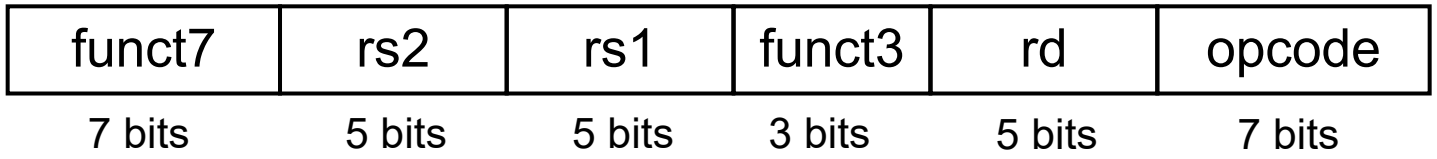
- 例：eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R型指令



- 指令字段
  - opcode：操作码
  - rd：目标寄存器号
  - funct3：3位功能码（扩展操作码）
  - rs1：第一个源操作数寄存器号
  - rs2：第二个源操作数寄存器号
  - funct7：7位功能码（扩展操作码）

R型指令的例子

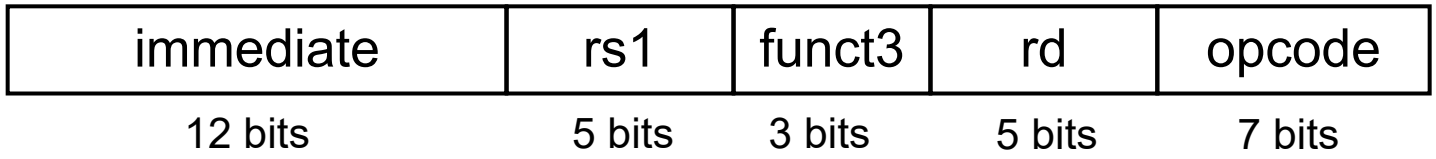



```
add x9,x20,x21
```

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

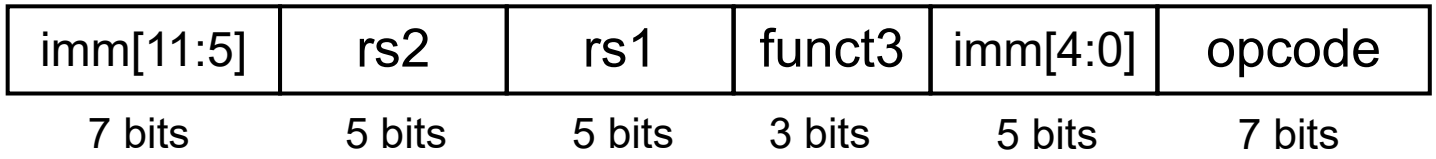
$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

RISC-V I型指令



- 立即数算术运算和取数指令
  - rs1：源或基地址寄存器号
  - immediate：常数操作数，或是加在基地址上的偏移量
- 设计原则3：好的设计需要好的折中
  - 不同的格式会使指令解码变复杂，但能一致使用32位的指令
  - 保持不同格式尽可能相似

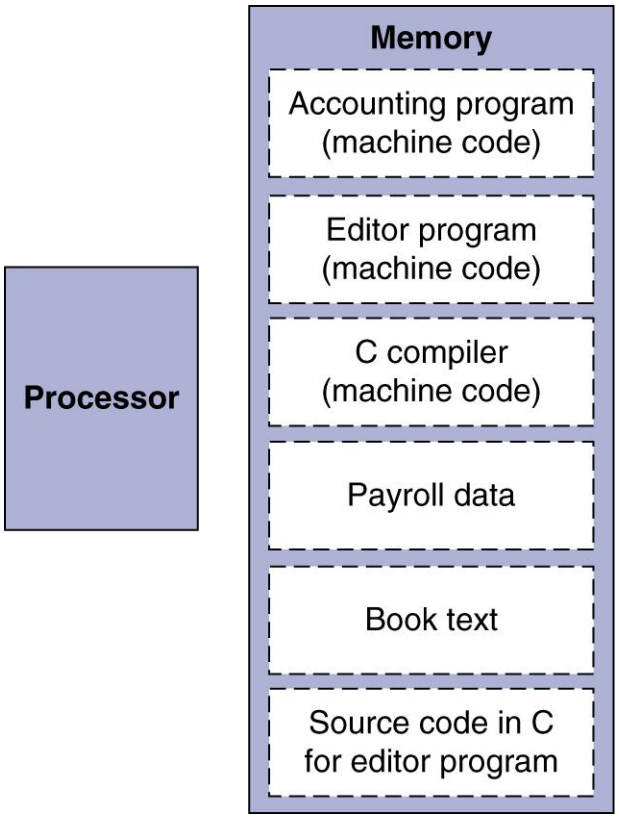
RISC-V S型指令



- 用于存数指令的另一种立即数格式
  - rs1：基地址寄存器号
  - rs2：源操作数寄存器号
  - immediate：加在基地址上的偏移量
    - 12位立即数拆分为7位和5位两个字段，是为了保持rs1和rs2字段位置不变

存储程序计算机

重点



- 指令像数据一样用二进制表示
- 指令和数据保存在存储器中
- 程序可以操作程序
  - 例如，编译器、链接器等等
- 二进制兼容使得编译后的程序能够在不同计算机上运行
  - 标准化的ISA

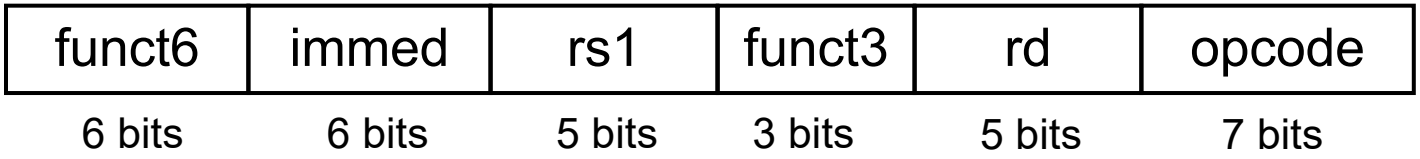
逻辑运算

- 按位操作的指令

操作	C	Java	RISC-V
左移	<<	<<	slli
右移	>>	>>>	srli
按位与	&	&	and, andi
按位或	|	|	or, ori
按位异或	^	^	xor, xori
按位取反	~	~	

- 用于对一个字提取或插入多组数位

移位运算

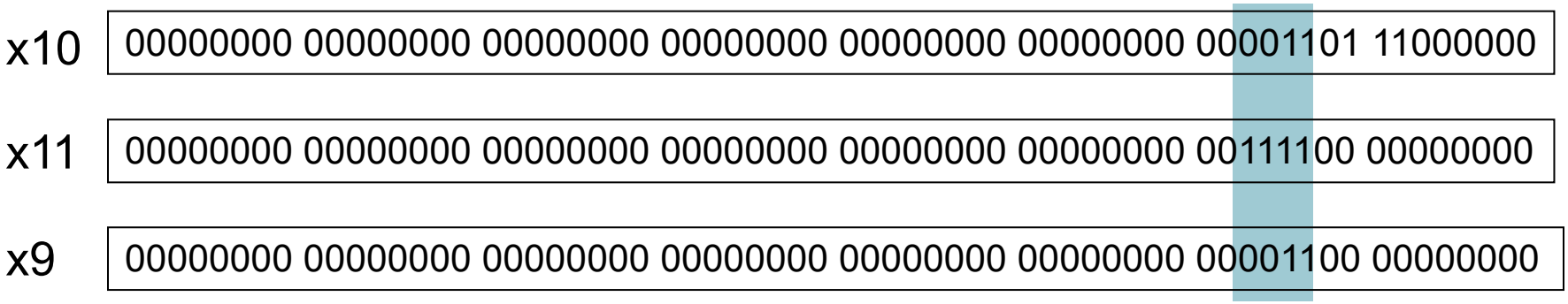


- immed：移动多少位
- 逻辑左移
  - 左移并以0填充空位
  - `slli`指令左移*i*位相当于乘以2^i
- 逻辑右移
  - 右移并以0填充空位
  - `srli`指令右移*i*位相当于除以2^i（仅适用于无符号数）

与运算

- 用于对一个字中的某些位进行掩码
  - 选出某些位，把其他位清0

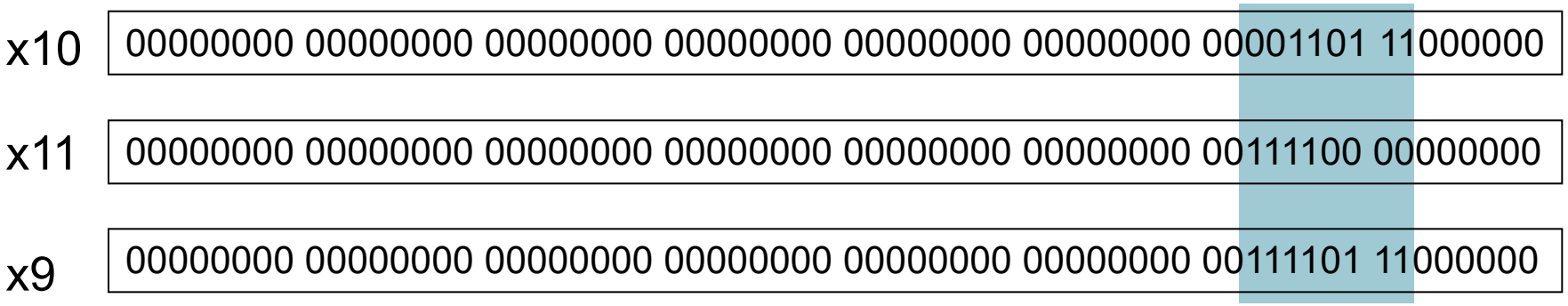
```
and x9,x10,x11
```



或运算

- 用于计入字中的某些位
  - 把某些位置1，保持其他位不变

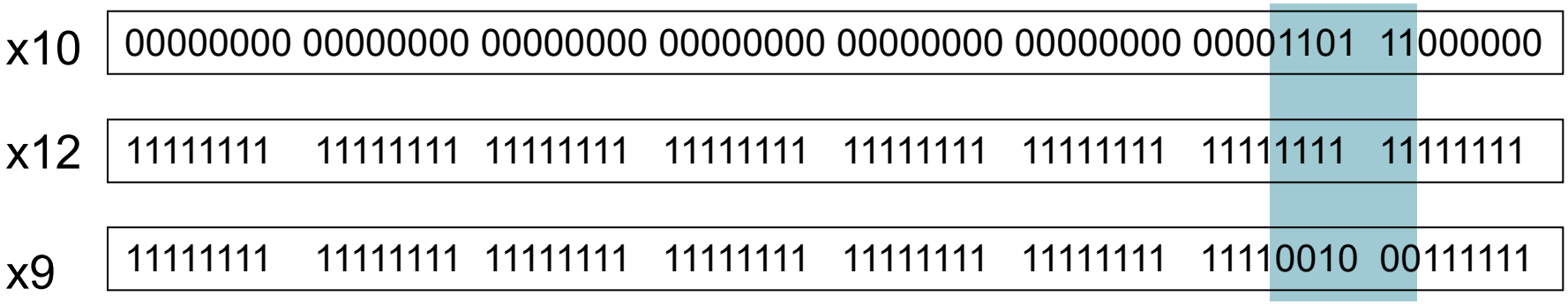
```
or x9,x10,x11
```



异或运算

- 差异运算
  - 把某些位置1，保持其他位不变

```
xor x9,x10,x12   # 按位取反运算
```



条件操作

- 如果条件为真则转到一个有标签的语句
  - 否则按顺序执行

- `beq rs1, rs2, L1`
  - 如果(rs1 == rs2)则转到标签为L1的语句

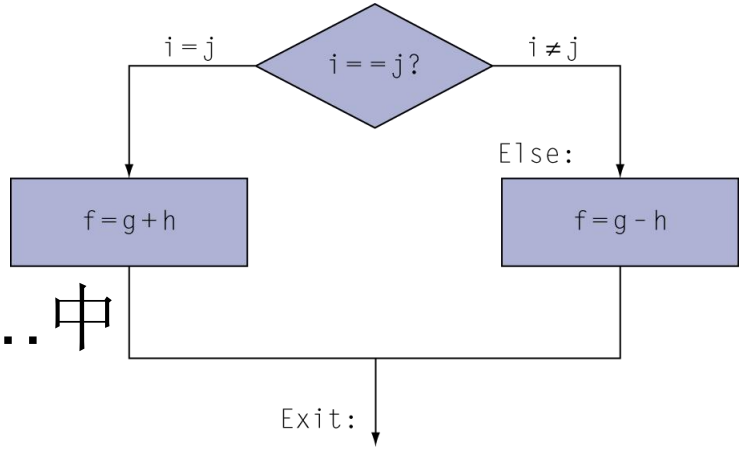
- `bne rs1, rs2, L1`
  - 如果(rs1 != rs2)则转到标签为L1的语句

编译if语句

- C代码：

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, …保存在x19, x20, …中



- 编译后的RISC-V代码：

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,x0,Exit   # 无条件跳转
Else: sub x19, x20, x21
Exit: …
```

由汇编器计算出地址

编译循环语句

- C代码：

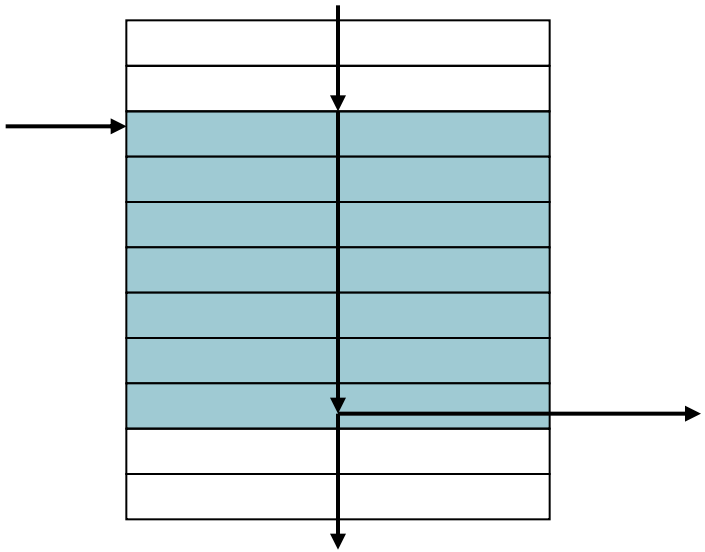
```
while (save[i] == k) i += 1;
```

  - i在x22中，k在x24中，save的地址在x25中
- 编译后的RISC-V代码：

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
       addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

基本块

- 基本块是这样的指令序列
  - 没有嵌入分支（除非在末尾）
  - 没有分支目标（除非在开头）



- 编译器可以识别基本块以进行优化
- 先进的处理器能够加速基本块的执行

更多的条件操作

- `blt rs1, rs2, L1`
  - 如果(rs1 < rs2) 则转到标签为L1的语句
- `bge rs1, rs2, L1`
  - 如果 (rs1 >= rs2) 则转到标签为L1的语句
- 例子
  - if (a > b) a += 1;
  - a保存在x22，b保存在x23

```
bge  x23, x22, Exit      # 当b >= a时跳转
addi x22, x22, 1
Exit:
```

有符号与无符号

- 有符号数比较：blt, bge
- 无符号数比较：bltu, bgeu
- 例
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - 若为有符号数，x22 < x23
    - –1 < +1
  - 若为无符号数，x22 > x23
    - +4,294,967,295 > +1

过程调用

- 所需步骤
  1. 将参数放入寄存器x10~x17
  2. 将控制转交给过程
  3. 获得过程所需存储空间
  4. 执行过程中的操作
  5. 为调用者将结果放入寄存器
  6. 返回调用位置（x1中保存的地址）

过程调用指令

- 过程调用：跳转和链接

  `jal x1, ProcedureLabel`

  - 将下一条指令的地址保存在x1中
  - 跳转到目标地址
- 过程返回：寄存器跳转和链接

  `jalr x0, 0(x1)`

  - 类似jal，但跳转到0 + x1中保存的地址
  - 把x0用作目的寄存器（实际x0不会被改变）
  - 同样可用于跳转到计算出的位置
    - 例如用于case/switch语句

叶过程的例子

- C代码：

```
long long int leaf_example (
    long long int g, long long int h,
    long long int i, long long int j) {
  long long int f;
  f = (g + h) - (i + j);
  return f; }
```

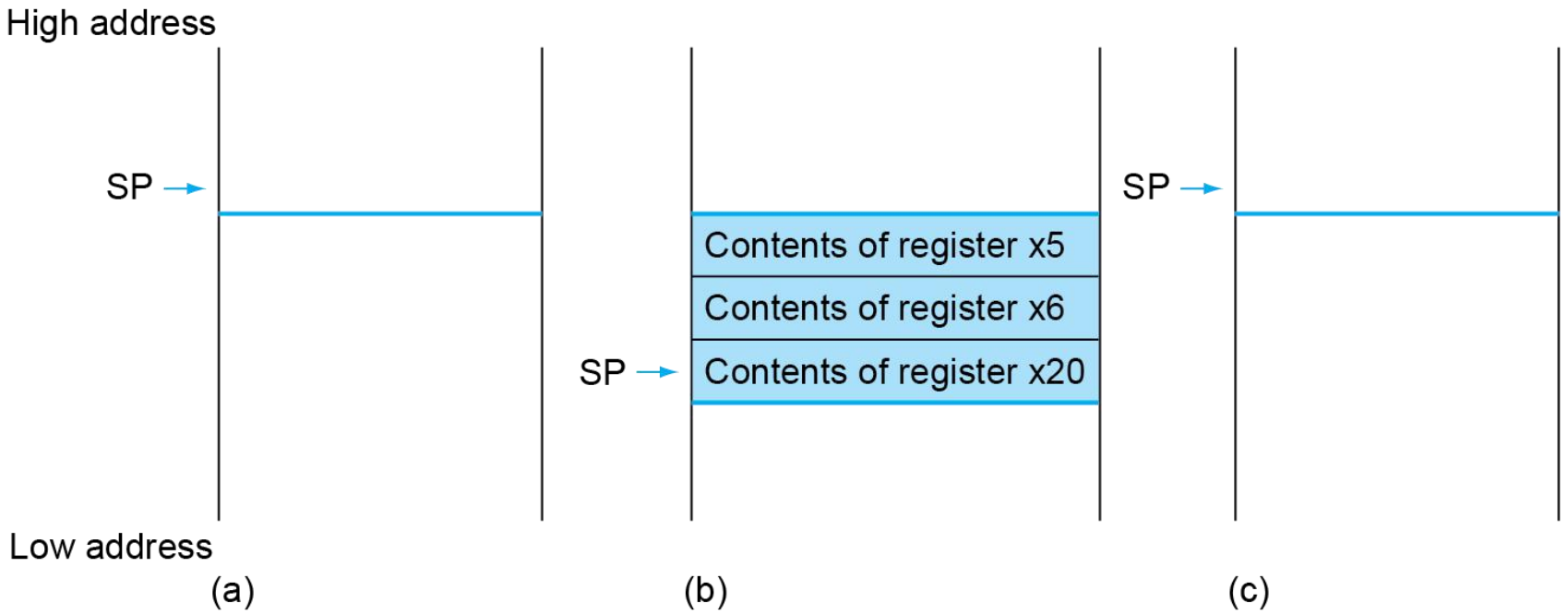
  - 参数g, …, j保存在x10, …, x13中
  - f保存在x20中
  - x5, x6保存临时变量
  - 需要把x5, x6, x20存入栈中

叶过程的例子

- RISC-V code:

```
leaf_example:
   addi sp,sp,-24          # 把x5, x6, x20的值存到栈中
   sd   x5,16(sp)
   sd   x6,8(sp)
   sd   x20,0(sp)
   add  x5,x10,x11         # x5 = g + h
   add  x6,x12,x13         # x6 = i + j
   sub  x20,x5,x6          # f = x5 – x6
   addi x10,x20,0          # 将f复制到返值寄存器
   ld   x20,0(sp)          # 从栈中恢复x5, x6, x20的值
   ld   x6,8(sp)
   ld   x5,16(sp)
   addi sp,sp,24
   jalr x0,0(x1)           # 返回到调用者
```

栈中的局部数据


High address
SP
Low address
(a)
Contents of register x5
Contents of register x6
SP
Contents of register x20
(b)
SP
(c)

寄存器的用法

- x5 – x7, x28 – x31: 临时寄存器
  - 被调用者不需要保留其中的值
- x8 – x9, x18 – x27: 需保存的寄存器
  - 如用到这些寄存器，被调用者需先保存原值，用完再恢复原值

非叶过程

- 调用其他过程的过程
- 对于嵌套过程，调用者需要在栈中保存：
  - 它的返回地址
  - 调用后还需要的任何参数和临时变量
- 调用后从栈中恢复

非叶过程的例子

- C代码：

```
long long int fact (long long int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

  - 参数n在x10中
  - 结果在x10中

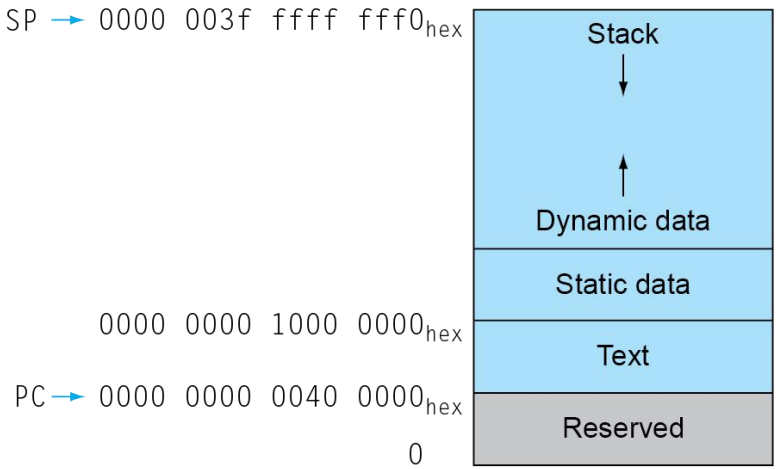
非叶过程的例子

- RISC-V code:

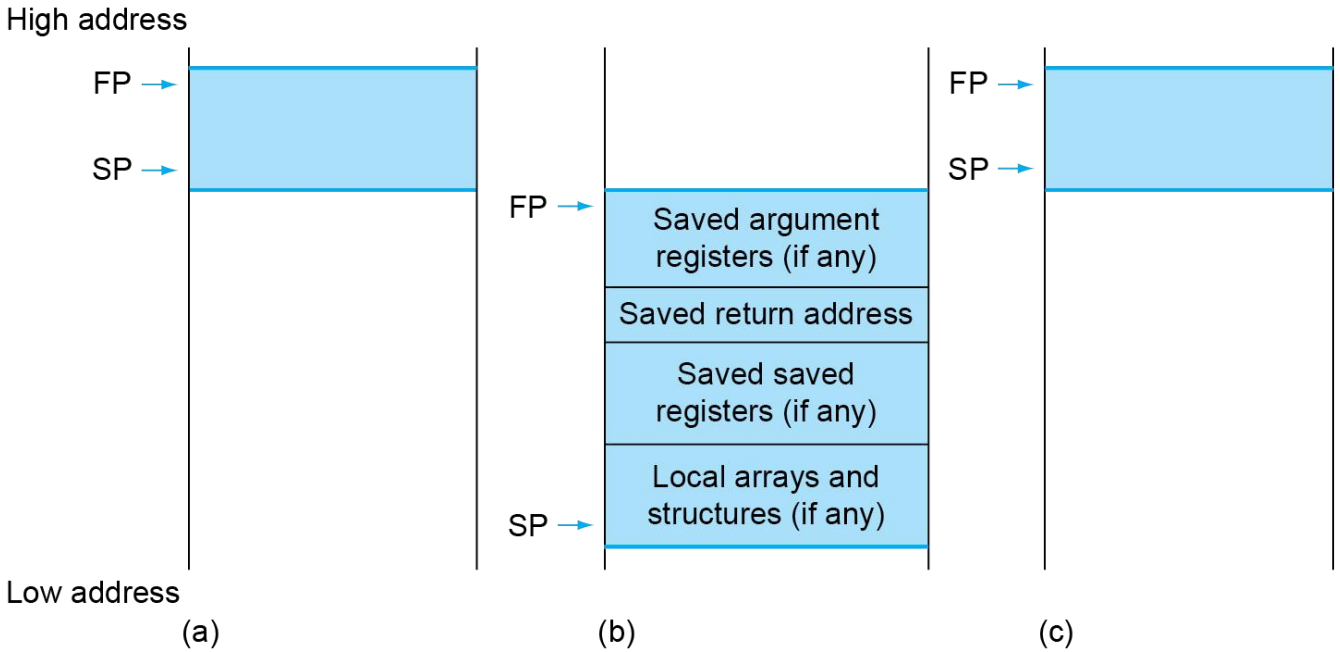
```
fact:
    addi sp,sp,-16          # 将返回地址和n保存到栈中
    sd   x1,8(sp)
    sd   x10,0(sp)
    addi x5,x10,-1          # x5 = n - 1
    bge  x5,x0,L1           # 若n >= 1，则跳转到L1
    addi x10,x0,1           # 否则，将返值置1
    addi sp,sp,16           # 出栈，此处不需要恢复原值
    jalr x0,0(x1)           # 返回
L1: addi x10,x10,-1         # n = n - 1
    jal  x1,fact            # 调用fact(n-1)
    addi x6,x10,0           # 将fact(n - 1)的结果存到x6
    ld   x10,0(sp)          # 恢复调用者的n
    ld   x1,8(sp)           # 恢复调用者的返回地址
    addi sp,sp,16           # 出栈
    mul  x10,x10,x6         # 返回n * fact(n-1)
    jalr x0,0(x1)           # 返回
```

内存布局

- 代码：程序代码
- 静态数据：全局变量
  - 例如C中的静态变量、常数组和字符串
  - x3（全局指针）初始化为适当的地址，加上正负偏移量可以访问这段内存空间
- 动态数据：堆
  - 例如C中的malloc、Java中的new
- 栈：自动存储



栈中的局部数据



- 由被调用者分配的局部数据
  - 例如，C的自动变量
- 过程帧（活动记录）
  - 被某些编译器用来管理栈存储

字符数据

- 按字节编码的字符集
  - ASCII：128个字符
    - 95个可显示字符，33个控制字符
  - Latin-1：256个字符
    - ASCII另加96个可显示字符
- Unicode：32位字符集
  - 用于Java、C++ 的宽字符等等
  - 涵盖世界上大多数的字母表和符号
  - UTF-8、UTF-16：变长编码

字节/半字/字操作

- RISC-V字节/半字/字的取数/存数
  - 从内存读一个字节/半字/字：符号扩展为64位存入rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - 从内存读一个无符号的字节/半字/字：零扩展为64位存入rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - 将一个字节/半字/字存入内存：保存最右端的8/16/32位
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

复制字符串的例子

- C代码（初级）：
  - 以null字符结束的字符串

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - 用x10保存x[]，x11保存y[]，x19保存i

复制字符串的例子

- RISC-V代码:

```
strcpy:
     addi sp,sp,-8          # 调整栈，留出1个双字的空间
     sd   x19,0(sp)         # x19入栈
     add  x19,x0,x0         # i=0
L1:  add  x5,x19,x11        # x5 = y[i]的地址
     lbu  x6,0(x5)          # x6 = y[i]
     add  x7,x19,x10        # x7 = x[i]的地址
     sb   x6,0(x7)          # x[i] = y[i]
     beq  x6,x0,L2          # 若y[i] == 0，则退出
     addi x19,x19,1         # i = i + 1
     jal  x0,L1             # 下一次loop迭代
L2:  ld   x19,0(sp)         # 恢复x19原值
     addi sp,sp,8           # 从栈中弹出1个双字
     jalr x0,0(x1)          # 返回
```

32位常量

- 常量一般较小
  - 12位立即数就够用
- 对于偶尔用到的32位常量

```
lui rd, constant
```

  - 复制20位常量到rd[31:12]，符号扩展rd[63:32]
  - 将rd[11:0]清0

```
lui x19, 976  # 0x003D0
```

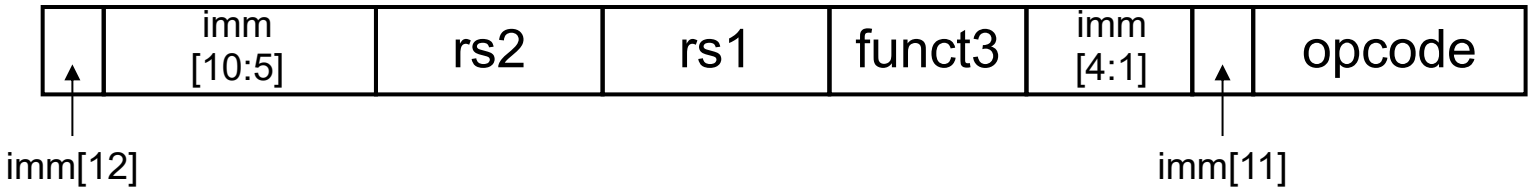
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

```
addi x19,x19,1280  # 0x500
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

分支寻址

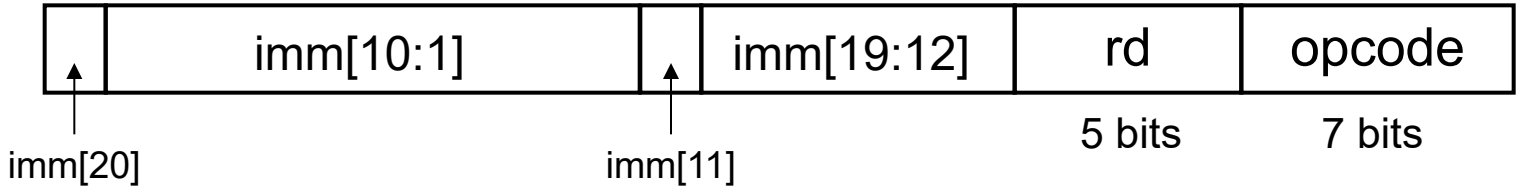
- 分支指令指定
  - 操作码、两个寄存器、目标地址
- 多数分支是近分支
  - 向前或向后
- SB格式：



- PC相对寻址
  - 目标地址 = PC + 立即数 × 2

跳转寻址

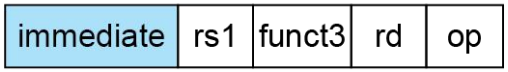
- jal的目标地址使用20位的立即数来实现更大范围的跳转
- UJ（无条件跳转）格式：



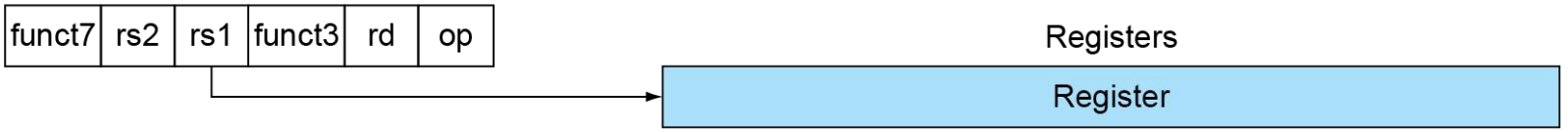
- 对于长跳转，例如32位的绝对地址
  - lui：将address[31:12]写入临时寄存器rs1
  - jalr rd, rs1, imm　# 加上address[11:0]，然后跳转到目标地址

RISC-V寻址模式总结

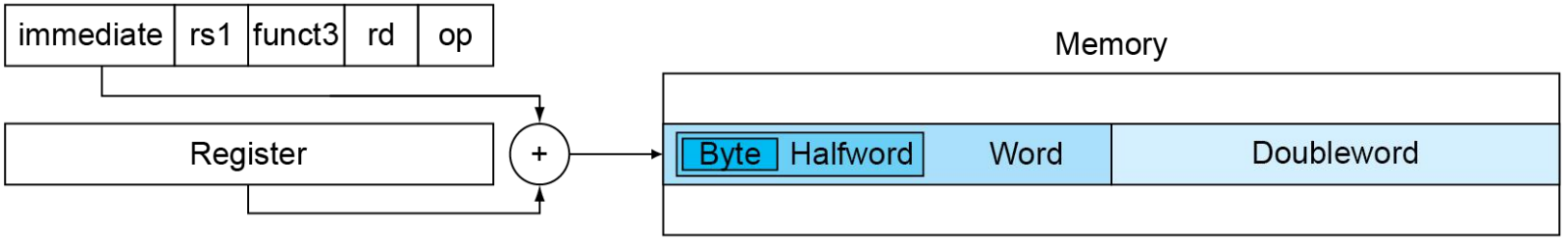
1. Immediate addressing



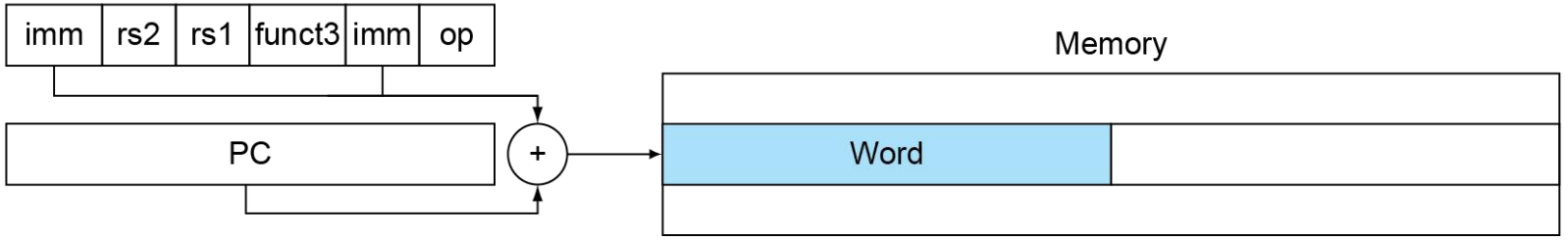
2. Register addressing



3. Base addressing



4. PC-relative addressing



RISC-V指令编码格式总结

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

同步

- 两个处理器共享存储器中的某一位置
  - P1写，然后P2读
  - 如果P1和P2不同步，则发生数据竞争
    - 结果取决于访问顺序
- 需要硬件支持
  - 原子读/写存储器的操作
  - 读写之间不允许有其他对这个位置的访问
- 可以是单一指令
  - 例如寄存器↔存储器的原子交换
  - 或者采用原子指令对

RISC-V中的同步

- 预留取数：`lr.d rd,(rs1)`
  - 从地址rs1处取数，保存到rd
  - 对内存地址设置预留
- 条件存数：`sc.d rd,(rs1),rs2`
  - 将rs2的内容保存到地址rs1
  - 如果从lr.d之后该位置没有被更改则执行成功
    - 在rd中返回0
  - 如果该位置被更改则执行失败
    - 在rd中返回非0值

RISC-V中的同步

- 例1：原子交换（用于检测/设置锁变量）

```
again:  lr.d x10,(x20)
        sc.d x11,(x20),x23 # x11 = 执行状态
        bne  x11,x0,again  # 如果存数失败则跳转
        addi x23,x10,0     # x23 = 从(x20)读到的数值
```

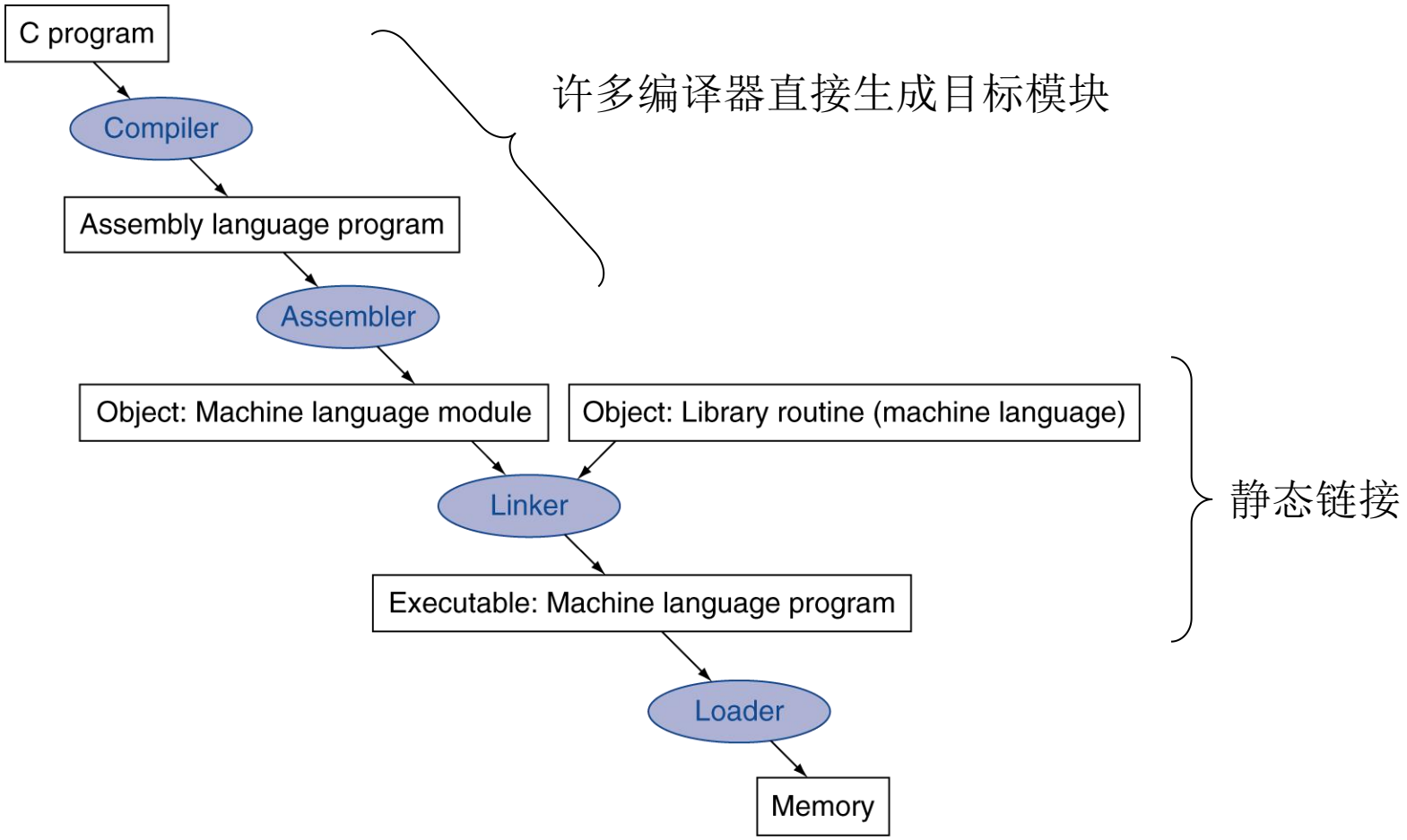
- 例2：加锁

```
        addi x12,x0,1          # x12 = 1，用于表示关锁
again:  lr.d x10,(x20)         # 读(x20)的锁值
        bne  x10,x0,again      # 如果锁不为0则跳转
        sc.d x11,(x20),x12     # 尝试向(x20)写1以关锁
        bne  x11,x0,again      # 如果存数失败则跳转
```

  - 解锁

```
        sd   x0,0(x20)         # 向(x20)写0以开锁
```

翻译和启动


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
许多编译器直接生成目标模块
静态链接

生成一个目标模块

- 汇编器（或编译器） 将程序翻译成机器指令
- 提供从各部分构建完整程序所需的信息
  - 头：描述目标模块的内容
  - 代码段：翻译后的指令
  - 静态数据段：分配的数据，作用于程序生命周期
  - 重定位信息：一些依赖于程序加载的绝对地址的内容
  - 符号表：全局定义和外部引用
  - 调试信息：用于关联到源代码

链接目标模块

- 生成可执行映像
  1. 合并各段
  2. 解析标签（确定它们的地址）
  3. 修补有位置依赖的引用以及外部引用
- 位置依赖性可留待重定位加载器来解决
  - 不过有虚拟内存就不需要这样做
  - 在虚拟内存空间中，程序能被加载到绝对位置

加载一个程序

- 将映像文件从磁盘加载到内存
  1. 读取文件头来确定各段大小
  2. 创建虚拟地址空间
  3. 将代码和初始化的数据复制到内存中
     - 或设置页表项来处理缺页
  4. 在栈上建立参数
  5. 初始化寄存器（包括sp、fp、gp）
  6. 跳转到启动例程
     - 将参数复制到x10等等并调用main函数
     - 当main函数返回时，进行exit系统调用

动态链接

- 仅在调用时链接/加载库过程
  - 要求过程代码是可重定位的
  - 避免了因静态链接到所有被引用的库而导致的映像膨胀
  - 自动调用新版本的库

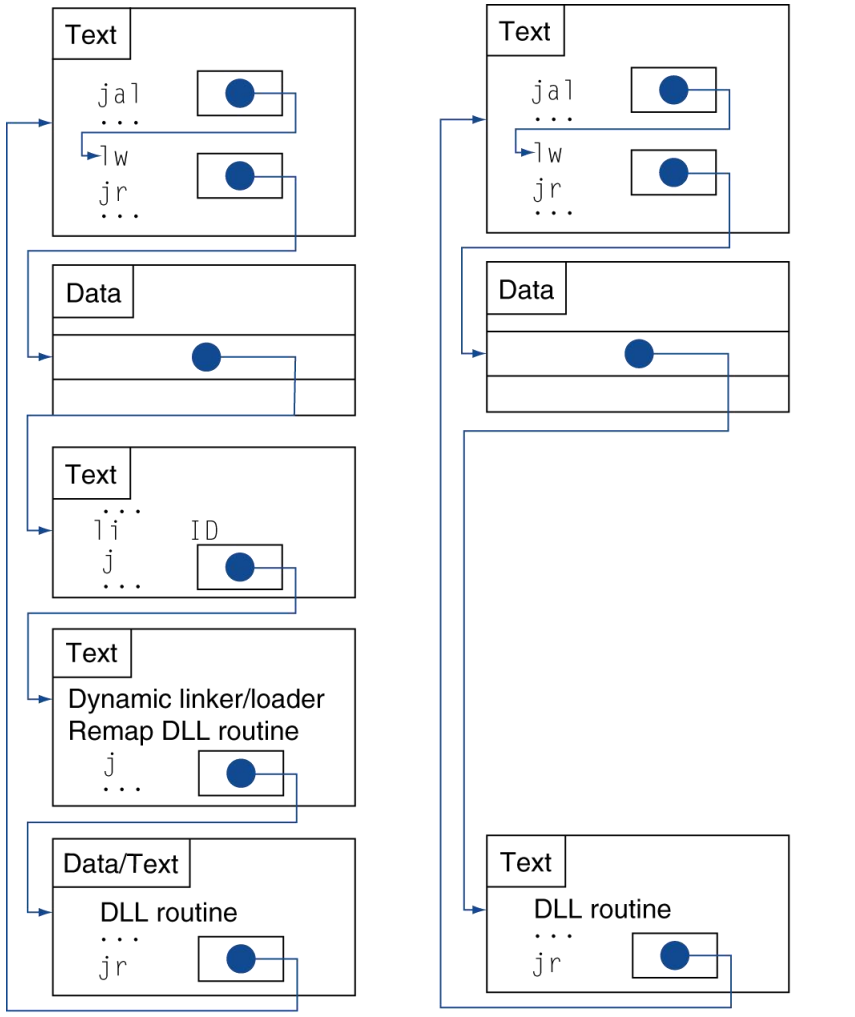
晚过程链接

间接表

桩函数：加载例程ID，
跳转到链接器/加载器

链接器/加载器代码

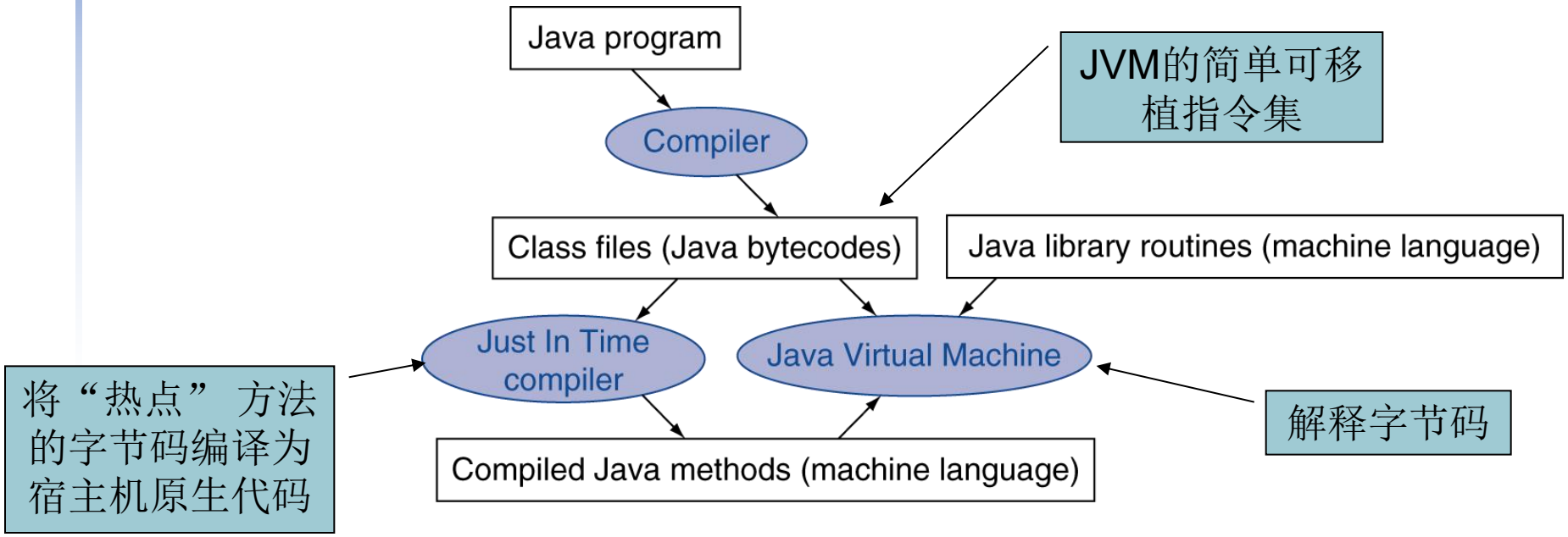
动态映射的代码



a. First call to DLL routine

b. Subsequent calls to DLL routine

启动Java应用程序


Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)
JVM的简单可移植指令集
将“热点”方法的字节码编译为宿主机原生代码
解释字节码

C排序的例子

- C冒泡排序函数的汇编指令示例
- swap过程（叶过程）

```
void swap(long long int v[],
          long long int k)
{
  long long int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v保存在x10，k保存在x11，temp保存在x5

swap过程

```
swap:
   slli x6,x11,3     # x6 = k * 8
   add  x6,x10,x6    # x6 = v + (k * 8)
   ld   x5,0(x6)     # x5（临时变量） = v[k]
   ld   x7,8(x6)     # x7 = v[k + 1]
   sd   x7,0(x6)     # v[k] = x7
   sd   x5,8(x6)     # v[k+1] = x5（临时变量）
   jalr x0,0(x1)     # 返回调用函数
```

C版本的sort过程

- 非叶过程（调用了swap）

```
void sort (long long int v[], size_t n)
{
  size_t i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v保存在x10，n保存在x11，i保存在x19，j保存在x20

外层循环

- 外层循环的骨架:
  - for (i = 0; i <n; i += 1) {

```
    li   x19,0              # i = 0
for1tst:
    bge  x19,x11,exit1      # 如果x19 ≥ x11（i≥n），跳转到exit1

    （外层循环的函数体）

    addi x19,x19,1          # i += 1
    j    for1tst            # 跳转到外层循环的判断语句
exit1:
```

内层循环

- 内层循环的骨架：
  - for (j = i − 1; j >= 0 && v[j] > v[j + 1]; j − = 1) {

```
    addi x20,x19,-1     # j = i -1
for2tst:
    blt  x20,x0,exit2   # 如果x20 < 0 (j < 0)，跳转到exit2
    slli x5,x20,3       # x5 = j * 8
    add  x5,x10,x5      # x5 = v + (j * 8)
    ld   x6,0(x5)       # x6 = v[j]
    ld   x7,8(x5)       # x7 = v[j + 1]
    ble  x6,x7,exit2    # 如果x6 ≤ x7，跳转到exit2
    mv   x21, x10       # 将参数x10复制到x21
    mv   x22, x11       # 将参数x11复制到x22
    mv   x10, x21       # swap的第一个参数是v
    mv   x11, x20       # swap的第二个参数是j
    jal  x1,swap        # 调用swap
    addi x20,x20,-1     # j -= 1
    j    for2tst        # 跳转到内层循环的判断语句
exit2:
```

维持寄存器值

- 将需保存的寄存器的值入栈：

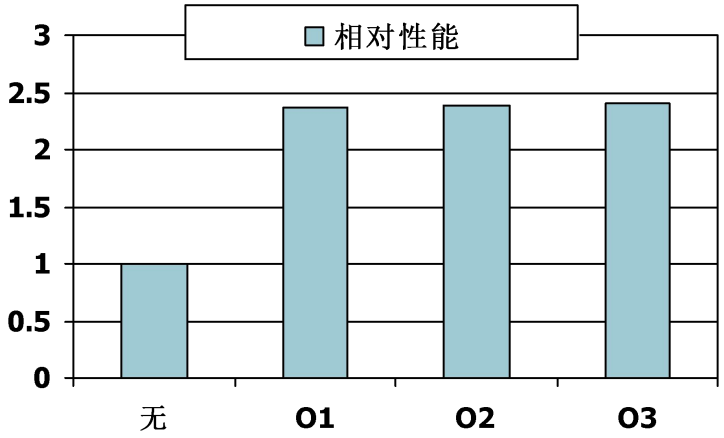
```
addi sp,sp,-40   # 在栈中留出5个寄存器（双字）的空间
sd   x1,32(sp)   # x1入栈
sd   x22,24(sp)
sd   x21,16(sp)
sd   x20,8(sp)
sd   x19,0(sp)   # x19入栈
```

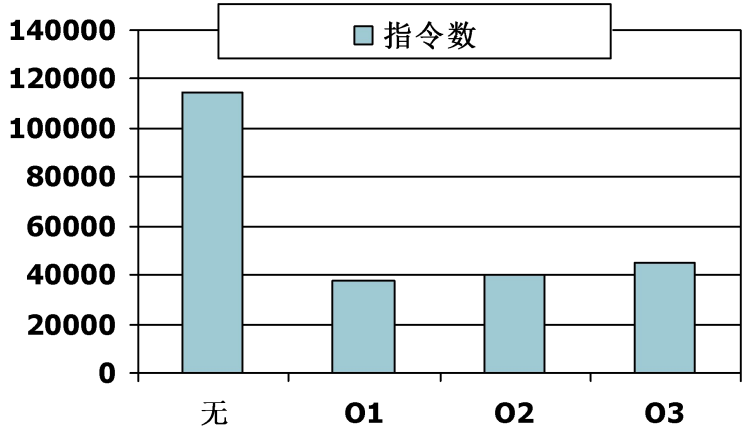
- 恢复需保存的寄存器的原值：

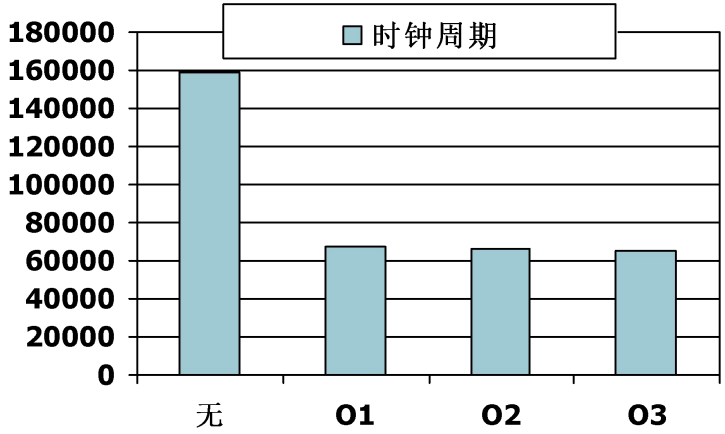
```
exit1:
  ld   x19,0(sp)   # 从栈恢复x19（出栈）
  ld   x20,8(sp)
  ld   x21,16(sp)
  ld   x22,24(sp)
  ld   x1,32(sp)   # 从栈恢复x1（出栈）
  addi sp,sp, 40   # 恢复栈指针
  jalr x0,0(x1)
```

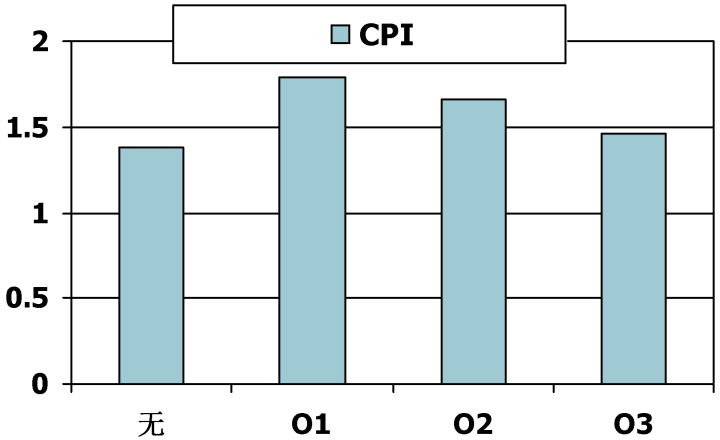
编译器优化的影响

用gcc for Pentium 4在Linux下编译

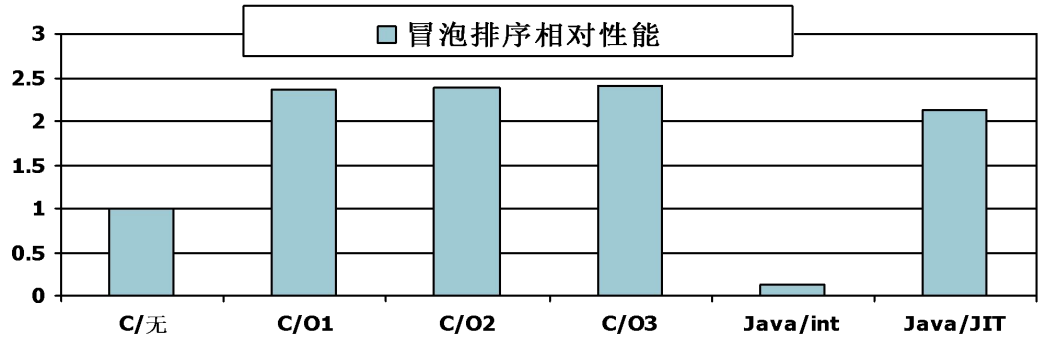


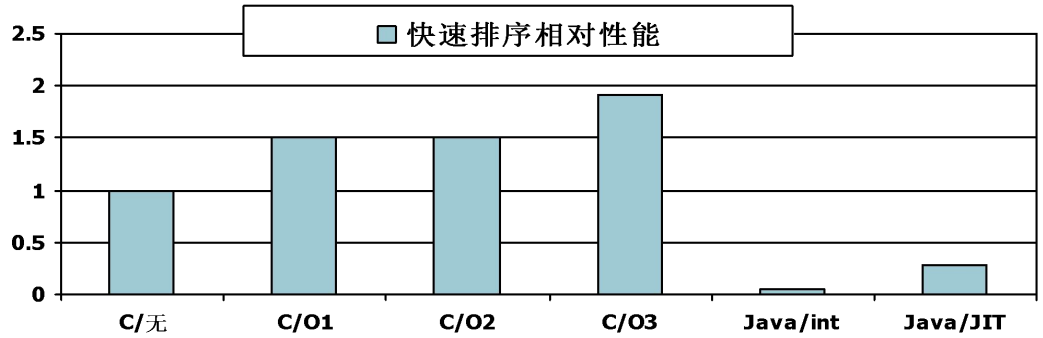


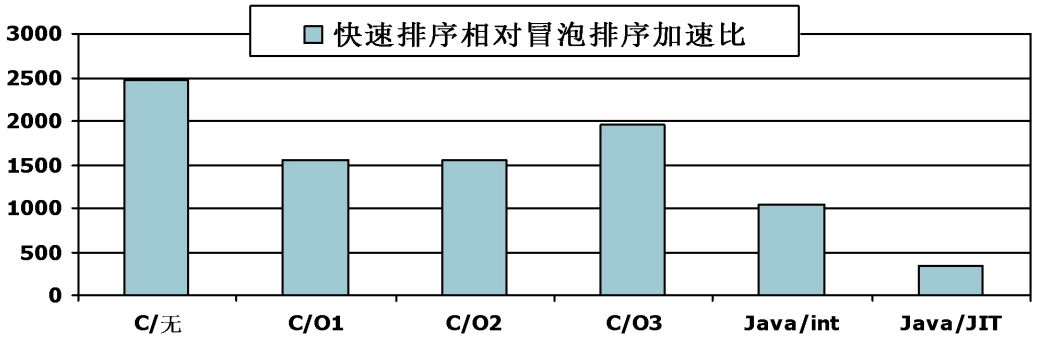




编程语言和算法的影响


冒泡排序相对性能
3
2.5
2
1.5
1
0.5
0
C/无
C/O1
C/O2
C/O3
Java/int
Java/JIT


快速排序相对性能
2.5
2
1.5
1
0.5
0
C/无
C/O1
C/O2
C/O3
Java/int
Java/JIT


快速排序相对冒泡排序加速比
3000
2500
2000
1500
1000
500
0
C/无
C/O1
C/O2
C/O3
Java/int
Java/JIT

经验

- 单看指令数或CPI都不是好的性能指标
- 编译器优化对算法敏感
- Java即时编译器生成的代码明显快于用JVM解释的代码
  - 有些情况下可以和优化过的C代码相媲美
- 没有什么可以弥补拙劣的算法！

数组与指针

- 数组索引包括
  - 用元素长度乘以下标
  - 与数组基址相加
- 指针直接对应存储器地址
  - 可以避免索引的复杂性

例：将数组清零

```
clear1(long long int array[], int size)
{
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
   li   x5,0       # i = 0
loop1:
   slli x6,x5,3    # x6 = i * 8
   add  x7,x10,x6  # x7 = array[i]的地址
   sd   x0,0(x7)   # array[i] = 0
   addi x5,x5,1    # i = i + 1
   blt  x5,x11,loop1  # if (i<size)
                      # go to loop1
```

```
clear2(long long int *array, int size) {
  long long int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
   mv x5,x10       # p = array[0]的地址
   slli x6,x11,3   # x6 = size * 8
   add x7,x10,x6   # x7= array[size]的地址
loop2:
   sd x0,0(x5)     # Memory[p] = 0
   addi x5,x5,8    # p = p + 8
   bltu x5,x7,loop2
                   # if (p<&array[size])
                   # go to loop2
```

数组与指针的比较

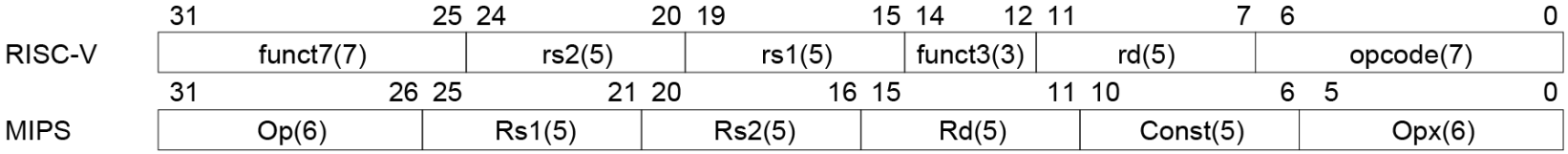
- 把乘“强度减少”为移位
- 数组版本要求移位在循环内部
  - 由于i的递增，需要有进行下标计算的部分
  - 对比指针的递增
- 编译器能与手动使用指针一样有效
  - 变量消除
  - 尽量让程序更加清晰和安全

MIPS指令

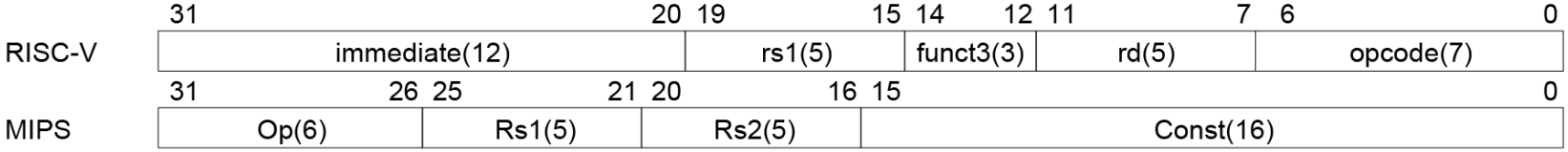
- MIPS：RISC-V商业化的前身
- 相近的基本指令集
  - 32位指令
  - 32个通用寄存器，寄存器0总是0
  - 32个浮点数寄存器
  - 只能用load/store指令访问内存
    - 对所有数据长度的寻址模式一致
- 条件分支不同
  - 对于<, <=, >, >=
  - RISC-V: blt, bge, bltu, bgeu
  - MIPS: slt, sltu（小于则置1，结果为0或1）
    - 然后用beq, bne来完成分支

指令编码

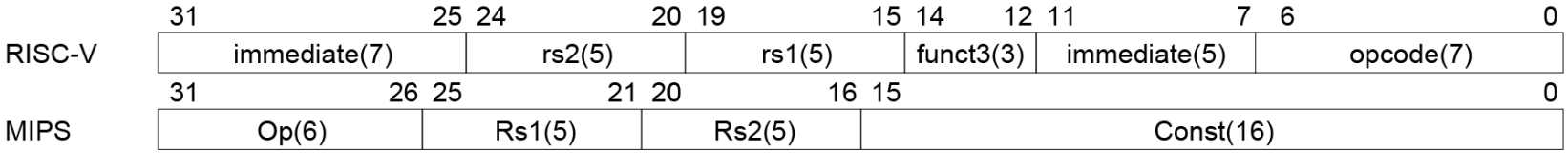
Register-register



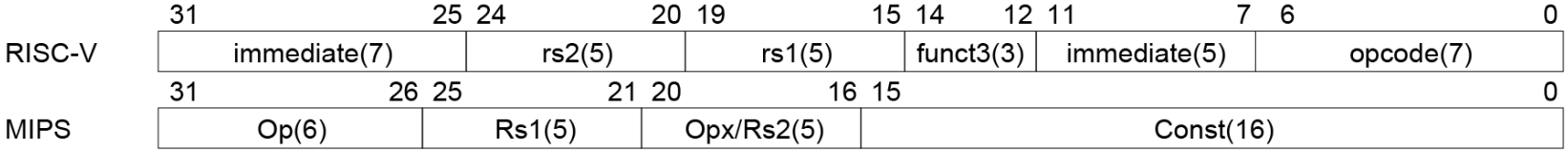
Load



Store



Branch



Intel x86指令集体系结构

- 演化时向后兼容
  - 8080 (1974): 8位微处理器
    - 累加器，外加3个索引寄存器对
  - 8086 (1978): 对8080的16位扩展
    - 复杂指令集 (Complex Instruction Set Computer, CISC)
  - 8087 (1980): 浮点数协处理器
    - 增加了浮点数指令和寄存器栈
  - 80286 (1982): 24位地址，内存管理单元(MMU)
    - 分段的内存映射和保护
  - 80386 (1985): 32位扩展（如今的IA-32）
    - 额外的寻址模式和操作
    - 分页的内存映射和分段

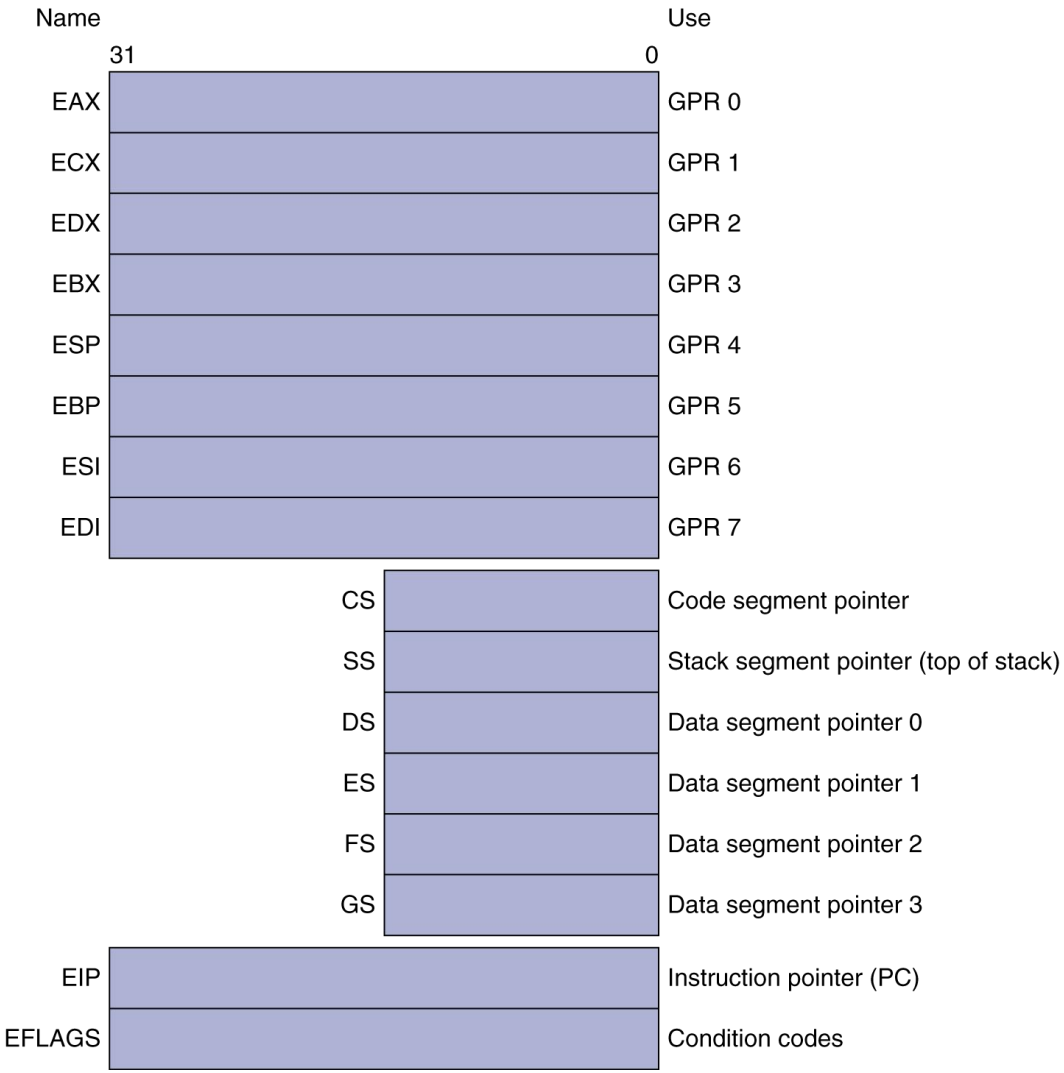
Intel x86指令集体系结构

- 进一步演化……
  - i486 (1989): 流水线式，片上cache和FPU
    - 兼容的竞争对手：AMD, Cyrix, …
  - Pentium (1993): 超标量，64位数据通路
    - 后期版本增加了MMX (Multi-Media eXtension，多媒体扩展)指令
    - 臭名昭著的FDIV缺陷
  - Pentium Pro (1995), Pentium II (1997)
    - 新的微架构（见Colwell, *The Pentium Chronicles*）
  - Pentium III (1999)
    - 增加了SSE (Streaming SIMD Extensions，流式单指令多数据扩展)和相应的寄存器
  - Pentium 4 (2001)
    - 新的微架构
    - 增加了SSE2指令

Intel x86指令集体系结构

- 再进一步……
  - AMD64 (2003): 将体系结构扩展为64位
  - EM64T – Extended Memory 64 Technology (2004)
    - Intel采纳了AMD64（有些改良）
    - 增加了SSE3指令
  - Intel Core (2006)
    - 增加了SSE4指令，支持虚拟机
  - AMD64（2007年发布）: SSE5指令
    - Intel拒绝跟随，而是……
  - Advanced Vector Extension（先进矢量扩展，2008年发布）
    - 更长的SSE寄存器，更多指令
- 如果Intel不保证兼容性，竞争对手会保证！
  - 技术优雅 ≠ 市场成功

基本的x86寄存器


Name
Use
31
0
EAX
GPR 0
ECX
GPR 1
EDX
GPR 2
EBX
GPR 3
ESP
GPR 4
EBP
GPR 5
ESI
GPR 6
EDI
GPR 7
CS
Code segment pointer
SS
Stack segment pointer (top of stack)
DS
Data segment pointer 0
ES
Data segment pointer 1
FS
Data segment pointer 2
GS
Data segment pointer 3
EIP
Instruction pointer (PC)
EFLAGS
Condition codes

基本的x86寻址模式

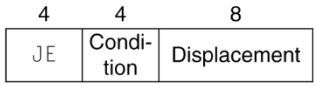
- 每条指令2个操作数

源/目的操作数	第二个源操作数
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

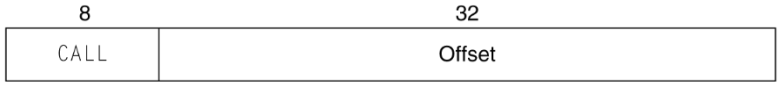
- 内存寻址模式
  - 寄存器中的地址
  - 地址 = R_{base} + 位移
  - 地址 = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - 地址 = $R_{base} + 2^{scale} \times R_{index}$ + 位移

x86指令编码

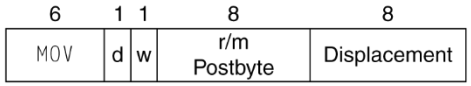
a. JE EIP + displacement



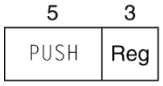
b. CALL



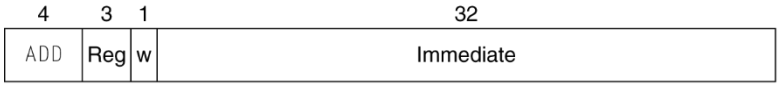
c. MOV EBX, [EDI + 45]



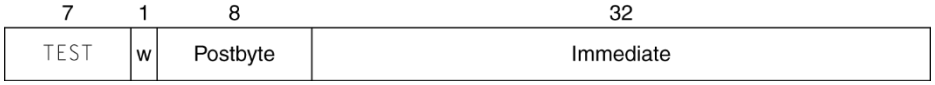
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- 变长编码
  - 前缀字节指定寻址模式
  - 前缀字节修饰操作
    - 操作数长度、重复次数、加锁……

实现IA-32

- 复杂指令集使得实现起来困难
  - 硬件将指令翻译成较简单的微操作
    - 简单指令：1-1
    - 复杂指令：1-许多
  - 微引擎类似RISC
  - 市场份额使这种处理在经济上行得通
- 性能比得上RISC
  - 编译器避免使用复杂指令

其他RISC-V指令

- 基础整数指令(RV64I)
  - 前面介绍的指令，外加
  - auipc rd, immed # rd = (imm<<12) + pc
    - 后面跟着jalr（加上12位立即数），用于长跳转
  - slt, sltu, slti, sltui: 小于则置1（类似MIPS）
  - addw, subw, addiw: 32位add/sub
  - sllw, srlw, sraw, slliw, srliw, sraiw: 32位移位
- 32位的变体：RV32I
  - 寄存器宽度为32位，操作为32位

指令集扩展

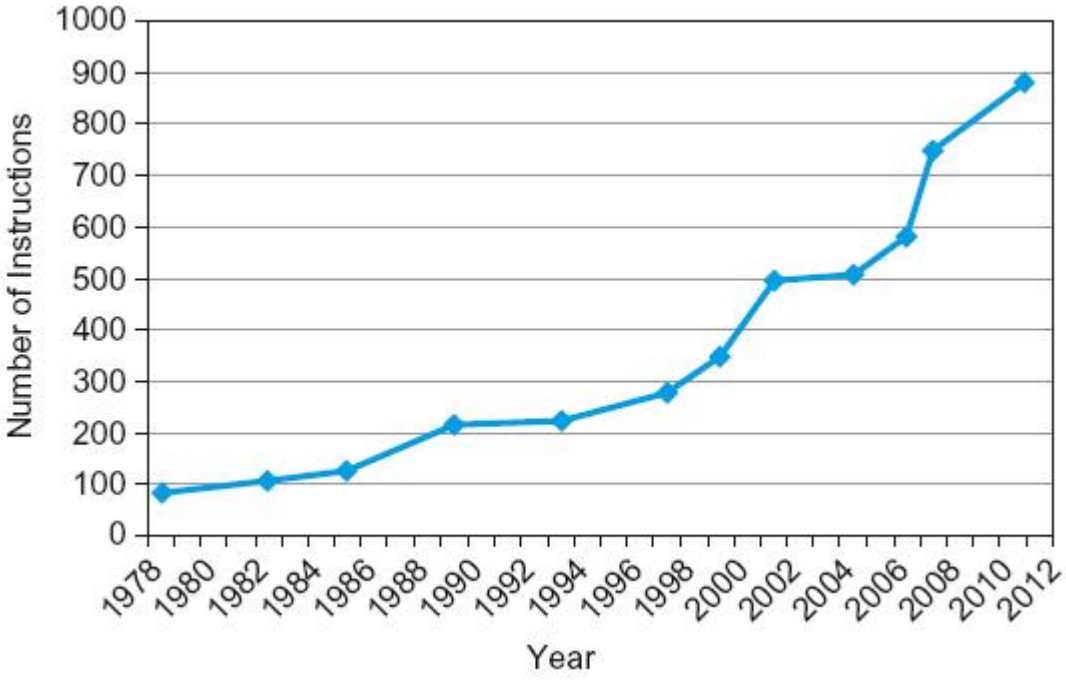
- M: 整型数乘、除、求余
- A: 存储器原子操作
- F: 单精度浮点数
- D: 双精度浮点数
- C: 压缩指令
  - 常用指令的16位编码

谬误

- 强大的指令 ⇒ 更高的性能
  - 需要的指令数更少
  - 但复杂的指令难以实现
    - 可能拖慢所有指令，包括简单指令
  - 编译器善于从简单指令生成快的代码
- 使用汇编代码来获得高性能
  - 但现代的编译器更善于配合现代的处理器
  - 更多的代码行数 ⇒ 更多的错误和更低的生产率

谬误

- 向后兼容性 ⇒ 指令集不需要改变
  - 但他们确实一直在加入更多的指令



x86指令集

陷阱

- 连续的字地址相差不是1
  - 按4递增而不是1！
- 在过程返回后，仍保留指向自动变量的指针
  - 例如经由一个参数传回指针
  - 指针在栈弹出后失效

本章小结

- 设计原则
  1. 简单源于规整
  2. 越小越快
  3. 好的设计需要好的折中
- 加速大概率事件
- 软件/硬件层次
  - 编译器、汇编器、硬件
- RISC-V：典型的RISC（精简指令集计算机）ISA
  - 对比x86